DSD Final Project Report

Chi-Tun Hsu (B11901053)
Wen-Tse Hsu (B11901070)
Hsuan-Ting Lin (B11901132)

June 15, 2025

1 Introduction

In this final project, we aimed to design and implement a pipelined RISC-V processor supporting the RV32I base integer instruction set. Our project extends the design of a single-cycle processor from HW2 to a five-stage pipeline to significantly improve throughput and performance.

To further enhance efficiency, we also integrated an instruction cache and a data cache, reducing memory access latency. Additionally, we extend our processor to support compressed instructions, multiplication instructions, and branch prediction.

The organization of the report is as follows:

- Section 2 introduces the **baseline processor** architecture, including the five-stage pipeline, hazard control mechanisms, and cache integration.
- Section 3 presents our **extensions** to the baseline design, covering support for compressed instructions (RVC), branch prediction strategies and performance analysis, as well as the design and integration of multiplication instructions.
- Section 4 discusses the **cache design**, including instruction and data caches and their impact on overall performance.
- Section 5 summarizes the **final optimization** and presents the final performance results.
- Section 6 concludes the report with a summary of our work.

2 Processor Architecture

In this section, we introduce the architecture of our baseline RISC-V processor, which supports the RV32I base instruction set and addresses pipeline hazards through a data forwarding unit and a stall control mechanism.

As shown in Figure 1, the RISC-V processor integrates hazard control and branch handling units in the ID stage. The hardware components are color-coded to indicate their functionality: orange blocks represent the control signal datapath, blue blocks represent the hazard control logic, and red blocks represent the data forwarding unit to the ID stage. For clarity, we do not explicitly show the full forwarding paths to the ID stage, as they would significantly increase diagram complexity.

We briefly discuss some designs of this baseline processor.

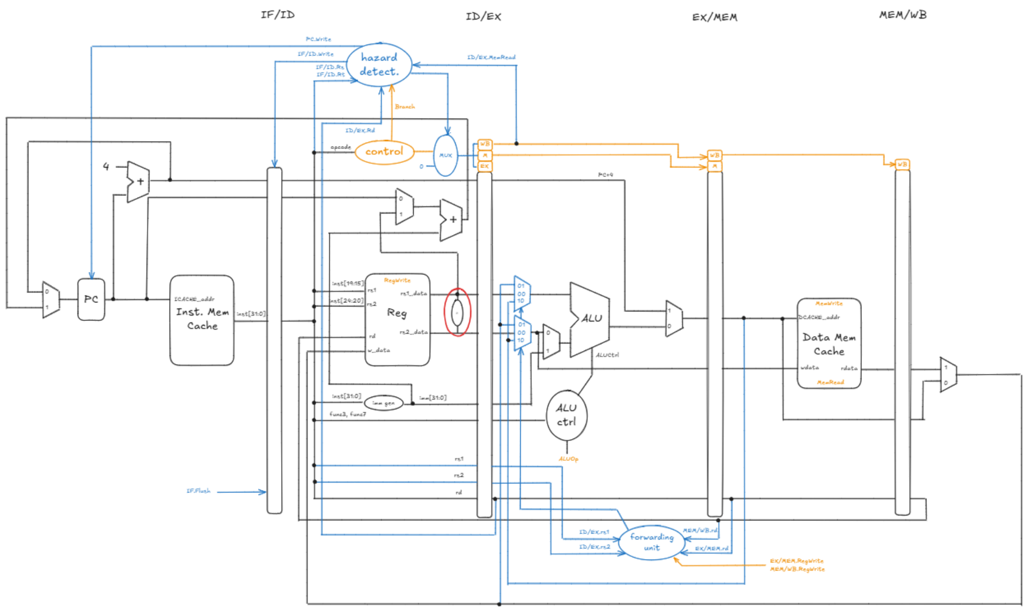

Figure 1: RISC-V Processor Architecture

2.1 Pipeline stages

The processor adopts a classic five-stage pipeline architecture, consisting of the **instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write-back (WB)** stages. Each stage is separated by positive-edge triggered pipeline registers, ensuring correct data propagation and synchronization between stages.

2.2 Hazard control

Two types of pipeline hazards are addressed in the baseline processor: **data hazards** and **control hazards**.

Data hazards are resolved using a data forwarding unit, which bypasses data from later pipeline stages to earlier ones, and a pipeline stall mechanism to handle load-use hazards where forwarding is insufficient.

Control hazards are managed by performing branch decisions in the ID stage; if a branch is taken, the processor flushes the IF stage to maintain correct instruction flow.

The hardware dedicated to hazard control are marked in blue in Figure 1, highlighting the data forwarding unit and the hazard control unit.

2.3 Branching

We moved the branch decision logic to the **ID stage** to minimize the number of cycles wasted on mispredicted branches. However, this creates a long critical path from the ID/EX pipeline registers through the ALU to the branch comparison logic. To avoid this long critical path when branch operands are still being computed in the EX stage, we insert a NOP into the EX stage and stall the IF and ID stages. In the following cycle, the correct operand is forwarded from the EX/MEM pipeline registers to complete the branch decision.

This additional stalling mechanism effectively shortens the critical path, allowing us to improve the processor's timing. As a result, the cycle time was successfully reduced **from 3.2 ns to 2.5 ns** after this optimization.

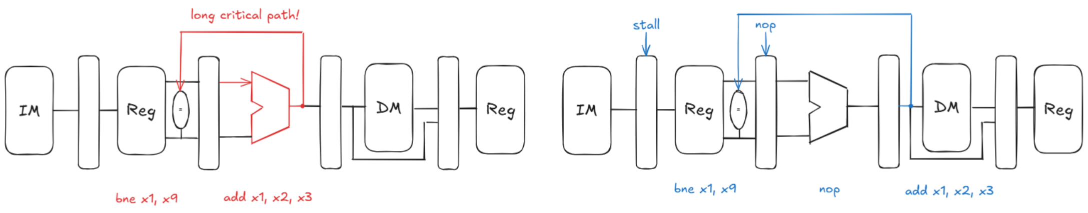

Figure 2: Stalling mechanism for branch decision

2.4 Cache

The instruction cache and data cache used in our baseline processor are derived from the design in HW3. Both caches are **2-way set-associative**, with a block size of 4 words and a total of 8 blocks per cache. The cache system is designed to reduce memory access latency by storing frequently accessed instructions and data in fast SRAM, enabling the processor to fetch instructions and data more efficiently compared to accessing the slower main memory.

3 Extensions

3.1 Compressed Instructions

One of the required extensions in this project is the support for compressed instructions, specifically a subset of the RISC-V Compressed (RVC) instruction set. By using compressed instructions, we can achieve higher instruction density, which improves memory utilization efficiency and can lead to better instruction cache performance.

However, having instructions aligned to half-words instead of full words introduces additional complexity in the instruction fetch stage.

First, we need to determine whether the current instruction is a compressed instruction in order to decide whether to increment the program counter (PC) by 2 or 4 bytes. Second, since cache addresses are aligned by words, we must implement an independent mechanism to correctly update the fetch address. Lastly, we need to handle cases where a jump or branch instruction is executed, potentially resulting in a PC that is aligned to the middle of a cache block. In such cases, if the fetched instruction is an uncompressed instruction, we need to stall for one cycle to fetch the remaining upper half of the instruction.

To address the issues mentioned above, we designed and implemented a **finite state machine (FSM)** to track the relationship between the current program counter (PC) and the instruction fetch address. The FSM updates its state and fetches instruction based on the current instruction type (RV32I or RVC) and handles state transitions during branch or jump instructions to ensure that subsequent instructions are correctly fetched from the cache. To ensure smooth transitions between states, we also maintain a **buffer** to store unused half-words that may be needed in the next cycle to fetch instructions and enable the address counter to increment in advance to prevent additional stalling.

We explain the finite state machine (FSM) shown in Figure 3 in detail by describing each state as follows:

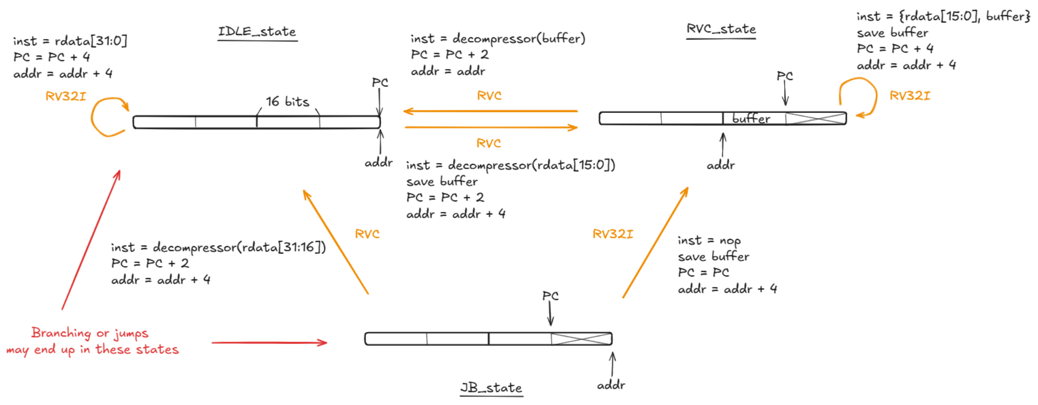

Figure 3: Finite state machine (FSM) for managing PC and instruction fetch address alignment

3.1.1 IDLE_state

The state is named IDLE_state because it is the initial state entered when the processor starts. In this state, both the program counter (PC) and the instruction fetch address are aligned to word boundaries.

If the fetched instruction is an RV32I instruction, the FSM reads the instruction as a full word from the cache. Both the PC and the instruction fetch address are incremented by 4 bytes, and the FSM remains in IDLE_state.

If the fetched instruction is an RVC instruction, the FSM reads the lower half-word and maps it to a 32-bit instruction using a decompressor. The PC is incremented by 2 bytes, and the instruction fetch address is incremented by 4 bytes. The FSM then transitions to RVC_state. Since the upper half-word of the cache word has not yet been used, it is stored in a buffer for later use.

3.1.2 RVC_state

The state is named RVC_state because it is used when the FSM first encounters an RVC instruction. In this state, the program counter (PC) is aligned to a half-word boundary, while the instruction fetch address is aligned to a word boundary, positioned 2 bytes ahead of the PC.

If the fetched instruction is an RV32I instruction, the FSM reads the lower half-word and concatenates it with the buffered upper half-word to form a 32-bit instruction. Both the PC and the instruction fetch address are incremented by 4 bytes, and the FSM remains in RVC_state.

If the fetched instruction is an RVC instruction, the FSM uses the buffered upper half-word as the complete compressed instruction and translates it into a 32-bit instruction using the decompressor. The PC is incremented by 2 bytes, thereby realigning the PC to a word boundary, and the FSM transitions back to IDLE_state.

3.1.3 JB_state

The state is named JB_state because it can only be accessed when a branch or jump instruction is executed, specifically if the target address is of the form $4n + 2$ bytes (i.e., aligned to a half-word boundary). If the branch or jump target address is aligned to a word boundary, the FSM will transition back to IDLE_state instead.

If the fetched instruction is an RV32I instruction, it is impossible to obtain the full instruction in one cycle because the upper half-word has not yet been read. In this case, the FSM writes the upper half-word into the buffer, stalls the PC while inserting a NOP, and increments the instruction fetch address by 4 bytes. After these operations, the FSM transitions to RVC_state, allowing the normal instruction fetch process to continue.

If the fetched instruction is an RVC instruction, the FSM reads the upper half-word from the cache and decompresses it into a 32-bit instruction. After this operation, the FSM transitions back to IDLE_state.

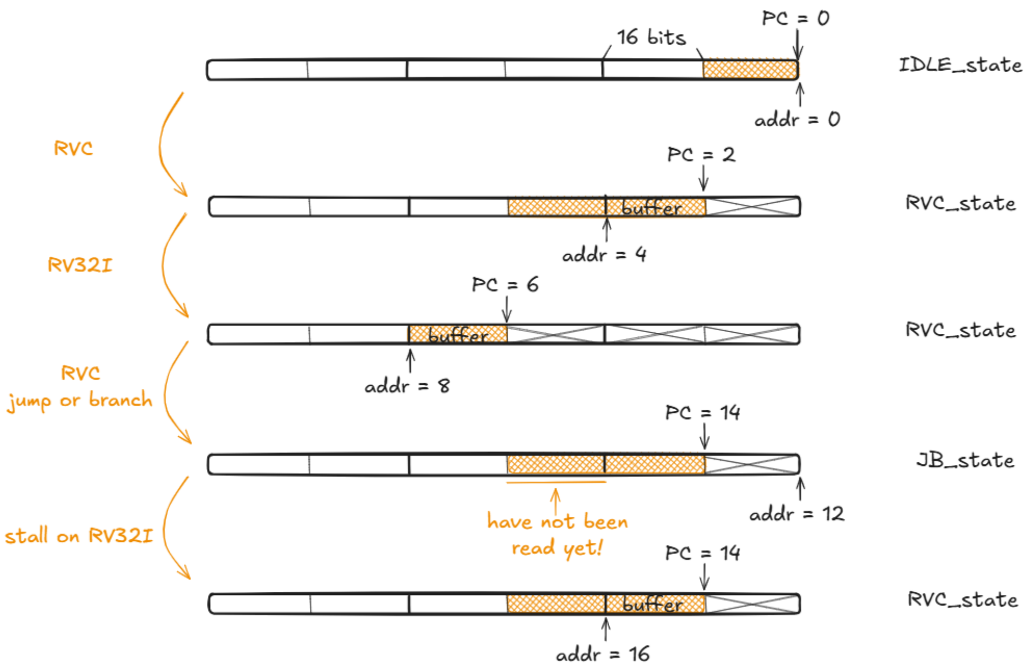

Figure 4: Example state transition of the FSM for instruction fetch

Figure 4 illustrates an example of the FSM's state transitions. During the example, the processor starts at PC = 0, and reads various types of instructions including a branch instruction that jumps to a PC in the form of $4n + 2$ bytes. The orange text with the arrows on the left indicates what type of instruction is currently being read, while the text on the right shows the current state of the FSM.

3.2 Branch Prediction

Branch predictors are commonly used in modern processors to **reduce the performance penalty** caused by flushing wrongly fetched instructions in the pipeline by predicting the outcome of branch instructions before they are resolved. A higher prediction accuracy leads to fewer flushed and wasted cycles. Additionally, mispredicting a branch may trigger an unnecessary instruction cache stall, fetching instructions that will not be used in the near future, thereby significantly increasing the performance penalty.

Extending our hardware to incorporate branch prediction is relatively straightforward. The baseline processor we implemented can be viewed as using an always not-taken strategy, where instructions are flushed if the branch outcome, computed in the ID stage, indicates that the branch is taken. To support more advanced prediction algorithms, we identify branch instructions and predict their outcome in the IF stage, flushing fetched instructions only when the prediction is incorrect.

For the two testcases used in the grading of our final project, we implemented various branch prediction strategies and reported their prediction accuracy in the simulation output. We describe these branch prediction mechanisms and present their performance on the two testcases in the following subsections, with detailed results provided in the corresponding tables.

3.2.1 Static Predictors

There are only two types of static branch predictors: **always not taken** and **always taken**. These types of predictors use minimal hardware resources and introduce the shortest critical path, making them simple and efficient to implement.

Table 1 shows the prediction accuracy of static branch predictors on the two testcases.

	QSort	Conv
Always taken	27.56	36.42
Always not taken	**72.44**	**63.58**

Table 1: Prediction accuracy (%) of static branch predictors

3.2.2 Global Saturation Counters

Global saturation counters are shared by all branches in the programs. Each branching result updates the n-bit saturation counter, which is used to predict the outcome of the next branch instruction. The saturation counter is incremented or decremented based on the branch outcome, and it saturates at its maximum or minimum value.

	QSort	Conv
1-bit global sat. counters	56.00	52.47
2-bit global sat. counters	70.27	50.00
3-bit global sat. counters	70.47	**63.58**
4-bit global sat. counters	**72.33**	63.58

Table 2: Prediction accuracy (%) of global saturation counters

3.2.3 PHT-based Predictors (PC-indexed, History-indexed, Gshare)

All three of the following branch prediction strategies use a common structure based on a Pattern History Table (PHT) containing n-bit saturating counters. The primary difference between these predictors lies in how the PHT is indexed:

- **PC-indexed**: The PHT is indexed using the lower bits of the program counter (PC).
- **History-indexed**: The PHT is indexed using a global branch history register (BHR), capturing the outcomes of recent branches.
- **Gshare**: The PHT is indexed by performing an XOR between the PC bits and the global branch history, allowing the predictor to capture both PC-local and global branch patterns.

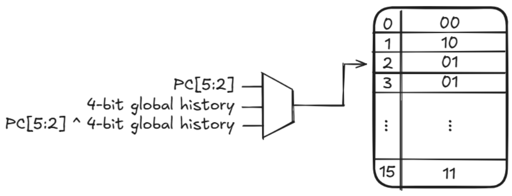

Figure 5: Illustration of the PHT-based predictors, showing the indexing methods and the PHT structure.

For each strategy, we evaluated the prediction accuracy under different PHT sizes (bit configurations). The results are shown in the following tables.

	4 entries		8 entries		16 entries		32 entries	
Counter Bits	Qsort	Conv	Qsort	Conv	Qsort	Conv	Qsort	Conv
1 bits	54.41	37.04	65.04	33.33	65.04	33.33	65.04	33.33
2 bits	68.03	57.41	73.05	**64.20**	74.11	64.20	74.11	64.20
3 bits	71.88	**63.58**	**76.07**	63.58	**75.86**	**63.58**	**75.86**	**63.58**
4 bits	**72.34**	63.58	75.37	63.58	75.06	63.58	75.86	63.58

Table 3: Prediction accuracy (%) of PC-indexed PHT predictor under different counter sizes and PHT entries

For **PC-indexed PHT predictors**, we observe that for each counter bit configuration, the prediction accuracy eventually saturates, and increasing the PHT size beyond a certain point does not lead to further improvements. This is because the limited number of distinct branch instructions in the program **prevents further aliasing** beyond a certain table size, resulting in unused PHT entries that do not contribute to improved accuracy.

	4 entries		8 entries		16 entries		32 entries	
Counter bits	Qsort	Conv	Qsort	Conv	Qsort	Conv	Qsort	Conv
2 bits	71.77	**64.20**	73.40	**78.40**	74.07	**82.10**	74.18	**83.33**
3 bits	**74.71**	64.20	**74.69**	75.93	**74.56**	79.01	**74.71**	79.63

Table 4: Prediction accuracy (%) of history-indexed PHT predictor under different history and counter sizes

For **history-indexed PHT predictors**, we observe that the prediction accuracy increases with the number of entries of the PHT. This indicates that previous branch results are useful for predicting future branches. However, as the counter bits increase, the performance of the predictors increases for the Qsort testcase and decreases for the Conv testcase. This may be due to the different branching nature of the two testcases. Further investigations show that the Conv testcase has a simpler branching pattern that can be predicted with fewer bits, while the Qsort testcase has more complex branching patterns that require more bits to accurately predict.

	4 entries		8 entries		16 entries		32 entries	
Counter bits	Qsort	Conv	Qsort	Conv	Qsort	Conv	Qsort	Conv
2 bits	75.22	61.73	74.53	74.07	75.29	**82.10**	75.71	**83.33**
3 bits	**75.58**	**66.05**	**75.35**	**75.93**	75.61	79.01	75.56	79.63
4 bits	74.35	63.58	74.79	73.46	**75.70**	74.07	**76.60**	74.69

Table 5: Prediction accuracy (%) of Gshare under different PHT and counter sizes

Gshare outperforms PC-indexed and history-indexed two-level predictors, as it **considers both PC and history** when finding the entry in the PHT table. Considering both PC and history reduces the aliasing problem seen in PC-indexed predictors, while also enabling more address-sensitive predictions compared to purely history-based predictors.

3.2.4 PAg and PAp Two-level Predictors

PAg and PAp are variants of general two-level predictors, first proposed by Yeh and Patt [1, 2]. In these predictors, we record the branch history of each PC and use it to index the PHT. The difference between PAg and PAp is that PAg uses a global PHT shared by all PCs, while PAp uses a separate

PHT for each branch. This significantly increases the hardware cost of PAp but can provide more accurate predictions in theory.

Table 6 and Table 7 show the prediction accuracy of PAg and PAp predictors under different PHT and counter sizes for the Qsort and Conv testcases.

	4 entries		8 entries		16 entries		32 entries	
Counter bits	Qsort	Conv	Qsort	Conv	Qsort	Conv	Qsort	Conv
2 bits	72.68	**88.89**	72.75	**93.83**	73.50	**94.44**	73.73	**3.83**
3 bits	**74.39**	88.27	73.97	89.51	75.25	90.12	75.28	90.12
4 bits	73.92	85.80	**75.09**	87.65	**75.27**	87.65	**75.32**	87.65

Table 6: Prediction accuracy (%) of PAg under different PHT and counter sizes

	4 entries		8 entries		16 entries		32 entries	
Counter bits	Qsort	Conv	Qsort	Conv	Qsort	Conv	Qsort	Conv
2 bits	76.04	**88.89**	75.58	**93.21**	75.50	**93.21**	75.66	**92.59**
3 bits	**76.68**	86.42	**76.29**	88.27	**76.15**	88.27	**76.38**	88.27
4 bits	75.92	81.48	76.18	83.33	76.07	83.33	76.11	83.33

Table 7: Prediction accuracy (%) of PAp under different PHT and counter sizes

The results show that PAp predictors excel in prediction accuracy on the QSort testcase, while PAg predictors perform best on the Conv testcase. One possible reason for this is that the Conv testcase has a shorter simulation time, so a large pattern history table (PHT) used by PAp may not be sufficiently trained before the simulation ends.

Branch prediction helps reduce the cycle count required to execute a program. However, it introduces **tradeoffs** between **prediction accuracy, latency, and hardware area**. The inclusion of branch prediction logic can increase the critical path delay, as the processor must identify branch instructions and determine the next PC, potentially prolonging the path from the PC to the instruction cache through the branch predictor. To make a design choice, we evaluate several promising predictors and analyze their AT scores to select an appropriate branch prediction strategy.

Branch Prediction	Cycle Time (ns)	Area (μm^2)	QSort Time (ns)	Conv Time (ns)	AT ($\times 10^{15}$)
Always Not Taken	2.5	432973	282913	54912.5	**6.726**
History-Indexed (16-entry, 2-bit Counter)	3.25	395952	340383	61281.47	8.259
GShare (16-entry, 2-bit Counter)	3.25	390206	340089	61281.47	8.132
PAp (4-entry, 2-bit Counter)	3.2	389702	340560	62716.8	8.324
PAp (8-entry, 2-bit Counter)	3.2	400316	340598.4	62681.6	8.546

Table 8: Comparison of branch predictors: cycle time, area, execution time (QSort, Conv), and AT score.

We discover that although the cycle count decreases with more accurate predictors, the branch prediction unit introduces a **longer critical path**, resulting in increased cycle time and total simulation time, which undermines the benefits of the reduced cycle count. According to the AT score, the Always Not Taken strategy offers a reasonable prediction accuracy while requiring a significantly shorter cycle time. We therefore choose to implement the **Always Not Taken** strategy in our final design.

3.3 Multiplication Instructions

In this project, we extended our processor to support multiplication instructions by adding a **3-stage pipelined multiplier**. This design choice was driven by the need to shorten the critical path, while enabling other instructions to execute concurrently and allowing successive multiplication instructions to proceed without stalling the pipeline.

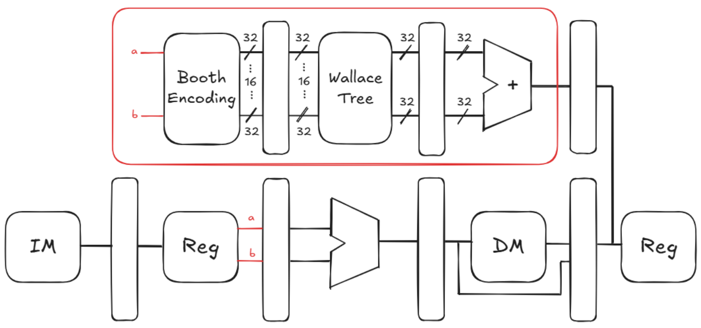

Figure 6: Illustration of the pipelined 3-stage multiplier, starting from the ID stage and completing in the MEM stage

We use one section to describe the data hazards introduced by the 3-stage multiplier and our solution to handle them, and another section to describe the design of the multiplier itself.

3.3.1 Data Hazards

The multiplier starts in the **ID stage**, reading operands from the register file and initiating the multiplication operation. However, the required data may still be computed in the **EX stage**, creating data hazards similar to those in branch verification. To resolve this, we **stall the IF and ID stages** until the EX stage completes its computation, following the method described in Section 2.3.

Figure 7 illustrates this kind of data hazard and the corresponding stalling and inserting of a NOP into the pipeline.

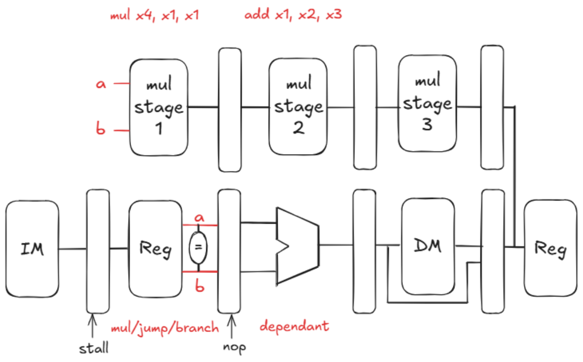

Figure 7: Data hazard introduced by the 3-stage multiplier, a mul/jump/branch operation with a data dependency with the previous instruction

The multiplier completes its operation at the end of the **MEM stage**, just like the load word instruction. Thus, this kind of ”load word hazard” will also be introdued if a multiplication instruction is followed by an instruction that needs the result of the multiplication.

This kind of data hazard and its corresponding solution are illustrated in Figure 8.

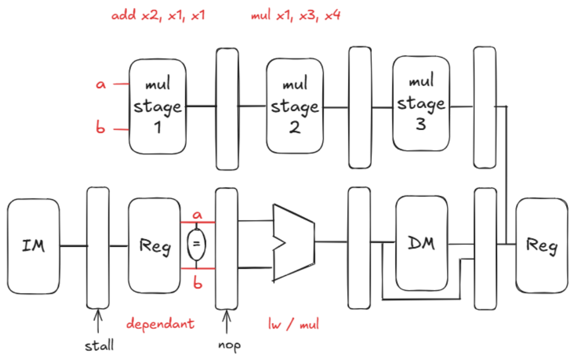

Figure 8: Data hazard introduced by the 3-stage multiplier, a mul/lw operation with a data dependency with the following instruction

The last type of data hazard introduced by the 3-stage multiplier occurs when an instruction that requires data at the beginning of the cycle in the ID stage depends on a result that will only become available at the end of the MEM stage.

While such instruction pairs will already have one NOP inserted (due to the earlier-stated data hazard condition), an additional NOP is required in these special cases to ensure correct timing and data flow. This prevents the consuming instruction from prematurely reading incorrect data.

Figure 9 illustrates this type of data hazard and the corresponding solution.

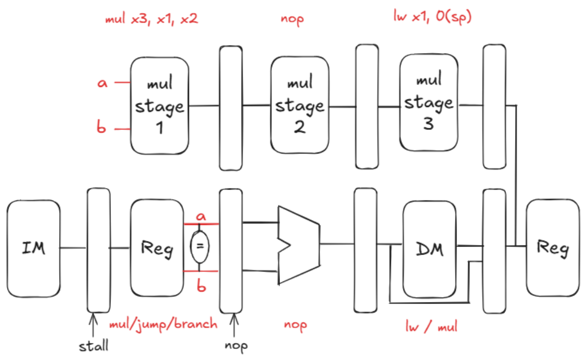

Figure 9: Data hazard introduced by the 3-stage multiplier, a mul/jump/branch in the ID stage requesting data that is still being calculated in the MEM stage

3.3.2 Multiplier Architecture

Inspired by the design of the pipelined multiplier introduced by Sun et al. [3], we implemented a 3-stage pipelined multiplier parallel to our processor pipeline to support 32-bit integer multiplication.

The first stage is the **partial product generation** stage, where the multiplier generates 16 partial products using **Booth encoding**. This technique effectively halves the number of partial products required for the multiplication, thereby reducing the overall complexity.

The second stage is the **Wallace Tree adder** stage, which compresses the 16 partial products into two 32-bit intermediate results: a carry and a sum. This fast reduction tree allows us to significantly shorten the critical path for this stage.

In the final stage, we use a simple **carry propagation adder** to sum the carry and sum results from the Wallace Tree, producing the final product.

This architecture balances latency across the three pipeline stages, ensuring that each stage completes its computation within a single cycle. Figure 10 illustrates the architecture of our pipelined multiplier (16-bit example). Since we only require the lower bits of the final result, we can safely truncate part of the higher bits in the partial products during the first stage without affecting correctness.

In the figure, green boxes indicate full adders, while orange boxes indicate half adders. Both types of adders generate a carry and sum output, which are then grouped and further reduced in the Wallace Tree until the final sum and carry results are produced at the end of the second stage.

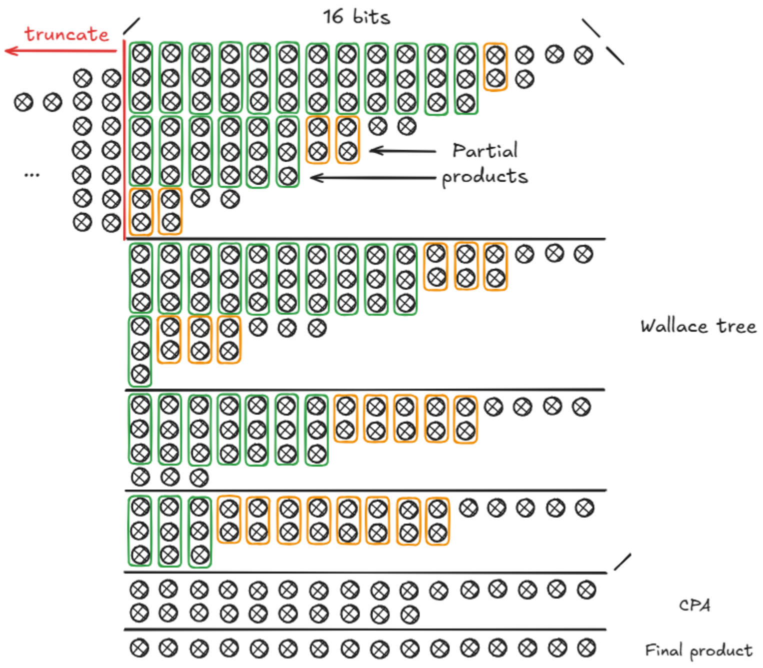

Figure 10: Architecture of the 3-stage pipelined multiplier (16-bit example)

4 Cache Design

For the instruction cache and data cache, several aspects must be considered, including cache size, associativity, block size (set to 4 words in this project), and replacement policy. We adopted **two-way associative** caches with **write-back schemes**, as this configuration strikes a balance between hit time and the ease of implementing a **least recently used (LRU)** replacement policy.

For the instruction cache, since it is never modified, we adopted a **read-only policy** to further simplify the design. This eliminates the need for write-back or write-through mechanisms, thereby reducing both area and access latency.

A higher block number (or larger cache size) reduces the miss rate for instruction caches and data caches, but increases the area and cycle time for the processor. To find an optimal tradeoff, we experimented with different cache configurations and selected the one that minimizes the AT score. The results of these experiments are summarized in Table 9.

Increasing block sizes significantly reduces the stalled cycles due to cache misses, and we further

ICache Blocks	DCache Blocks	Cycle Time (ns)	Area (μm^2)	AT ($\times 10^{15}$)
8	8	2.5	361731	8.225
16	8	2.5	432973	6.726
8	16	2.7	458481	8.456
16	16	2.7	498783	**5.905**

Table 9: Cache configuration and corresponding cycle time, area, and AT score.

investigate the impact of different cache sizes on the hit rate of caches. The impact of cache sizes on the hit rate is shown in Table 10 for Conv and Table 11 for QSort.

Block sizes	ICache hit rate	DCache hit rate
8	0.914	0.707
16	0.977	0.815

Table 10: Cache hit rates on Conv testcase.

Block sizes	ICache hit rate	DCache hit rate
8	0.949	0.916
16	0.979	0.950

Table 11: Cache hit rates on QSort testcase.

As demonstrated, larger cache configurations result in higher hit rates. Although this comes at the cost of increased area and slightly longer cycle times, the **net benefit in performance** justifies the choice. Therefore, we adopted a configuration with 16 blocks for both the instruction and data caches in our final design.

5 Final Optimization and Results

We optimized the AT score by fine-tuning the cycle time as well as the input and output delays, based on the optimal configurations derived in the previous sections. The final design employs an always not taken branch prediction strategy, along with a 16-block instruction cache and a 16-block data cache. The final performance results are summarized in Table 12.

Cycle Time (ns)	Area (μm^2)	QSort Time (ns)	Conv Time (ns)	AT ($\times 10^{15}$)
2.65	491680.76	270870.45	42419	**5.649**

Table 12: Final optimized performance results of our design.

6 Conclusion

For this final project, we successfully met the requirements of designing a pipelined RISC-V processor supporting the RV32I base integer instruction set for the checkpoint presentation. We then extended the design to support compressed instructions, multiplication instructions, and conducted a thorough study on the effects of different branch prediction strategies and cache configurations on our processor's performance.

We designed a finite state machine to handle instruction fetch address alignment for **compressed instructions**, enabling the processor to correctly fetch and decode both RV32I and RVC instructions while maintaining correct PC updates and instruction fetch address transitions.

Our work demonstrates that **branch prediction** can effectively reduce cycle count, but it also requires a longer cycle time to incorporate the additional branch prediction datapath. Overall, the best strategy is to adopt an always not taken predictor, which balances cycle time and prediction accuracy.

For **multiplication instructions**, we added a 3-stage pipelined multiplier to the processor. The multiplication operation is initiated in the ID stage and completed in the MEM stage. This design introduces additional data hazards, which we carefully analyzed and addressed by customizing the hazard control unit to handle these new data dependencies.

Last but not least, we optimized the cache configuration and carefully fine-tuned the synthesis parameters to maximize performance. As a result, the final AT score of our design reaches 5.649×10^{15} (μm^2 ns^2).

Work Assignment Table

Work Item	**Chi-Tun Hsu**	**Wen-Tse Hsu**	**Hsuan-Ting Lin**
Checkpoint processor			✓
RVC support	✓		✓
Multiplication		✓	
Branch prediction			✓
Cache design	✓	✓	✓
Report writing	✓	✓	✓
Testing & debugging	✓	✓	✓

Table 13: Work Assignment Table

References

[1] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th annual international symposium on Microarchitecture (MICRO-24)*. ACM, 1991, pp. 51–61.

[2] ——, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th annual international symposium on Computer Architecture (ISCA)*. ACM, 1992, pp. 124–134.

[3] R. Sun, H. Liu, R. Zhang, and J. Qu, "Design and implementation of RISC-V based pipelined multiplier," in *Journal of Physics: Conference Series*, vol. 2625. IOP Publishing, 2023, p. 012006, 4th International Conference on Electrical, Electronic Information and Communication Engineering, 21-23 April 2023, Dalian, China.